SDU

University of Southern Denmark

DM852

Heuristics & Approximations Algorithms

Submitted To:
Marco Chiarandini
Lene Monrad Favrholdt
IMADA
Mathematics & Computer
Science Department

Submitted By :
Alexander Lerche Falk
Narongrit Unwerawattana
Spring - Master of
Computer Science

Contents

1   Introduction

This project shows metaheuristic algorithms, resolving the Capacitated Vehicle Routing Problem (CVRP). The difference between metaheuristics and heuristics is in the solution part. For heuristics, you are trying your best to find a solution, even though it is not optimal. The algorithm is adapted to the problem in such a greedy approach, it can get stuck in a local optimum. This is fine since it is the idea of heuristics: "just solve the problem".
Metaheuristics are less greedy and tends to be more problem independent. They accept temporary solutions and allow "bad" steps as an attempt to get a better solution. You have a local- and global optimum to keep track of the best solution found. You can say metaheuristics are exploting heuristics to avoid getting trapped. In this metaheuristics implementation project, we have chosen to implement two algorithms for CVRP: Simulated Annealing (SA) and Ant Colony Optimization (ACO). The SA algorithm is the one we have uploaded for electronic submisison at http://valkyria.imada.sdu.dk/D0App/. The ACO algorithm is implemented but does not perform well. We will compare the algorithms with our previous heuristic / local search project and lastly, show our results of computation for our two algorithms.

    — Alexander & Narongrit

2   Simulated Annealing

The Simulated Annealing (SA) algorithm is inspired from the annealing process in metal, where you are altering the physical state of the metal by heating and cooling it. The inspiration can be used in computer science as well. We can use the algorithm on CVRP by starting off by generating a solution to the problem and not "care" about the initial solutions. The better steps we are taking, and better solutions we are finding, the more careful we are going to be in finding a solution. In the beginning we allow random and bad steps to be performed, while later, we make better calculations.

We have developed an algorithm to solve CVRP using Simulated Annealing technique by relating CVRP with existed problems i.e. Bin packing problem and Knapsack problem. Using this approach, we can guarantee maximum amount of vehicles necessary required with approximation analysis on bin packing hueristic algorithm. In this demonstration, we use well studied algorithm First-Fit-Decreasing which is guaranteed to use no more than $\frac{11}{9}\text{OPT} + 1$ number of required vehicles[1]. With combination of simple Traveling Salesman Problem's local search algorithm, 2-opt, we are able create the initial feasible solution as described in 2. Next step of Simulated Annealing involved creating new feasible solution by altering current solution. Which in CVRP case, is to perform points exchange action between routes. In order to allow as much possibility of exchanges as it could without breaking capacity constrain, we choose simple algorithm to find exchanges possibility based on Knapsack problem 3.

Algorithm 1 Simulated annealing with First-Fit-Decreasing and Knapsack combination

---

Require: *customers*: List of customer coordinate with it's capacity
Require: *route_cap(route)*: returns capacity of route
Require: *route_cost(route)*: returns capacity of route
Require: *tsp_local_search(route)*: performs travelling salesman local search and returns improved route
Require: *is_accept(best_cost, cost, temp)*: an acceptance criterion function returns boolean based on Metopolis condition
Require: *max_cap*: Maximum capacity for each route
Require: *depot*: Depot point

1: **function** SIMULATED_ANNEALING_FFD(*customers, temp, alpha, no_improve_limits*)
2:     *routes* ← CVRP_FIRST_FIT_DECREASING(*customers, depot*)
3:     *best_cost* ← total cost of *routes*
4:     *best_routes* ← *routes*
5:     *current_cost* ← *best_cost*
6:     *current_routes* ← *routes*
7:     *curr_no_improve_trying* ← 0
8:     **while** True **do**
9:         **if** *curr_no_improve_trying* == *no_improve_limits* **then** break
10:         *origin_route_id, target_route_id* ← randomly pick 2 distinct index of *routes*
11:         *origin_route_point* ← randomly pick element from *route[origin_route_id]*
12:         *upper_bound, lower_bound* ← maximum and minimum capacity such that solution still feasible after exchange action
13:         *target_neighbors_set* ← KNAPSACK_COMBINATION(
                route[target_route_id], lower_bound, upper_bound)
14:         **if** *len(target_neighbors_set)* == 0 **then** continue
15:         *target_neighbors* ← randomly pick an element from *target_neighbors_set*
16:         *new_route_i, new_route_i* ← exchange neighbors between two route and assign to new variables
17:         *new_route_i* ← *tsp_local_search(new_route_i)*
18:         *new_route_j* ← *tsp_local_search(new_route_j)*
19:         *old_tours_cost*   ←   *route_cost(route[origin_route_id])*   +
    route_cost(route[target_route_id])
20:         *new_tours_cost* ← *route_cost(new_route_i)* + *route_cost(new_route_j)*
21:         *new_cost* ← *current_cost* − *old_tours_cost* + *new_tours_cost*
22:         **if** *is_accept(best_cost, new_cost, temp)* **then**
23:             update *temp* with *alpha*
24:             update *current_routes* with new routes
25:             *curr_no_improve_trying* ← 0
26:             **if** *new_cost* < *best_cost* **then**
27:                 update best_cost and best_routes with new routes
28:         **else**
29:             *curr_no_improve_trying* ← *curr_no_improve_trying* + 1
        return *best_routes*

Algorithm 2 First-Fit-Decreasing for CVRP

Require: *customers*: List of customer coordinate with it's capacity
Require: *tsp_local_search(route)*: performs travelling salesman local search and returns improved route
Require: *max_cap*: Maximum capacity for each route
Require: *depot*: Depot point
Require: *route_cost(route)*: returns capacity of route
Ensure: *routes*: solution route

1: **function** CVRP_FIRST_FIT_DECREASING(*customers*)
2:    *sorted_capacity_points* ← sort customer points by it's capacity
3:    *routes* ← []
4:    add a route with *depot* to *routes*
5:    **for all** *point* in *sorted_capacity_points* **do**
6:       *is_fit* ← *False*
7:       **for all** *route* in *routes* **do**
8:          **if** *route_cost(route)* + *point.capacity* ≤ *max_cap* **then**
9:             add *point* to *route*
10:            *is_fit* ← *True*
11:            break
12:       **if** !*is_fit* **then**
13:          add new route with *depot* and *point* to *routes*
14:    **for all** *route* in *routes* **do**
15:       *route* ← *tsp_local_search(route)*
      return *routes*

Algorithm 3 Knapsack Combination

Require: *customers*: List of customer coordinate with it's capacity
Require: *max_cap*: Maximum capacity for each route
Require: *depot*: Depot point
Require: *max_points_amt*: Number of maximum combination points
Require: *route_cap(route)*: returns capacity of route
Ensure: *combination_set*: possible combination of points in route s.t. their capacity not exceeds *max_cap*

1: **function** KNAPSACK_COMBINATION(*route, lower_bound, upper_bound*)
2:     *points* ← randomly pick *max_points_amt* points from *route* but *depot*
3:     *combination_set* ← []
4:     **function** KNAPSACK_COMBINATION_RECUR(*current_point_id, current_capacity, knapsack*)
5:         **if** *current_point_id* == *len(points)* **then**
6:             **if** *route_cap(knapsack)* > *lower_bound* **then**
7:                 add knapsack to *combination_set*
8:         *knapsack_combination_recur(*
                current_point_id + 1, *current_capacity, knapsack*)
9:         **if** *current_cap* + *points[current_point_id]* ≤ *max_cap* **then**
10:             add *points[current_point_id]* to *knapsack*
11:             *current_cap* ← *points[current_point_id].capacity*
12:             *knapsack_combination_recur(*
                current_point_id + 1, *current_capacity, knapsack*)
13:     *knapsack_combination*(0, 0, [])
14:     **return** *combination_set*

---

instance	CHH		SA		instance	CHH		SA	
	k	cost	k	cost		k	cost	k	cost
A-n32-k05	5	867	5	980	CMT01	6	604	5	553
A-n33-k05	5	743	5	661	CMT02	11	920	10	894
A-n33-k06	6	766	6	816	CMT03	8	985	8	987
A-n34-k05	6	889	5	778	CMT04	12	1196	12	1568
A-n36-k05	5	862	5	807	CMT05	17	1496	17	3406
A-n37-k05	5	741	5	669	CMT11	8	1111	7	1517
A-n37-k06	7	1112	6	949	CMT12	10	909	10	1180
A-n38-k05	6	822	5	730	Golden_01	9	6055	9	8796
A-n39-k05	5	883	5	827	Golden_02	10	9121	10	14227
A-n39-k06	6	887	6	999	Golden_03	9	12144	9	17920
A-n44-k06	6	1029	6	939	Golden_04	10	15644	10	24338
A-n45-k06	7	1014	6	978	Golden_05	5	7330	5	10954
A-n45-k07	7	1250	7	1161	Golden_06	7	9698	7	14509
A-n46-k07	7	997	7	1039	Golden_07	9	11655	9	17513
A-n48-k07	7	1232	7	1128	Golden_08	10	12835	10	19443
A-n53-k07	8	1142	7	1054	Golden_09	15	590	14	653
A-n54-k07	8	1258	7	1217	Golden_10	16	763	16	843
A-n55-k09	9	1158	9	1300	Golden_11	18	963	18	1090
A-n60-k09	9	1412	9	1394	Golden_12	20	1192	19	1199
A-n61-k09	11	1300	9	1111	Golden_13	28	972	26	1175
A-n62-k08	8	1379	8	1363	Golden_14	31	1194	30	1471
A-n63-k09	10	1764	9	1676	Golden_15	35	1501	33	1743
A-n63-k10	11	1504	10	1442	Golden_16	39	1880	37	2092
A-n64-k09	9	1530	9	1501	Golden_17	22	774	23	1289
A-n65-k09	10	1274	9	1260	Golden_18	28	1108	29	2350
A-n69-k09	10	1297	9	1266	Golden_19	34	1578	35	3279
A-n80-k10	10	1903	10	1920	Golden_20	39	2084	40	4309

Table 1: Result comparing between Heuristic algorithm and Metahuristic running with time limit 240 seconds on an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz with 4 GB allocated RAM running Ubuntu 16.04.

3   Ant Colony Optimization

The Ant Colony Optimization (ACO) algorithm comes from the evolutionary algorithms, where computer science meets nature. In this algorithm, we have led us to be inspired by how ants find food in the nature. They are starting by spreading out in some area, randomly. When they seem to find trails, which could indicate to be good trails to find food, they leave pheromone behind them. The pheromone is used by other ants to determine their probability of taking a trail. If they are standing in a cross-section and they have to choose, they are taking the path with the highest pheromone. At some point in their exploration to find the best trail to obtain food, all the ants are using the same trails to get food and get back safe home.

We can apply the logic of the ants in the CVRP as well. By letting the instance the space of which the requests are "plotted" be the area to find a solution, we can start by sending one "ant" out to a random point. This is going to be our current point. From this point, we are going to calculate every probability of moving to the next point the one with the highest "score". We can calulate the probabilty of moving by the doing the following:

First we establish the initial pheromone levels from all the points to every counter other point:

This is stored in matrix: $M = \begin{bmatrix} 0 & 0.50 & 0.20 \\ 0.125 & 0 & 0.60 \\ 1.20 & 0.355 & 0 \end{bmatrix}$

We also keep track of the points being visited, so we do not keep visiting the same nodes. Then we start calculating a score for each point from the current point. We do this by the formular:

$$score = \alpha^{\frac{1.0}{distance(i,j)}} \beta \tag{1}$$

The αvalue defines the importance of the pheromone level, where the βvalue defines the importance of the distance between two points. After finding the score for each point from the current point, we can calculate the probability by dividing the score by the summation of all scores:

$$prob = \frac{score}{\sum_{i=1}^{n} score_i} \tag{2}$$

Where **n** is equivalent to the number of points. We can now append additional pheromone to the trail from the current point to the next point, by dividing:

$$newpheromone = \frac{1}{distance(i,j)} \tag{3}$$

Now we can continue with our usual CVRP properties, where we ensure the capacity of the vehicle is not breached. We continue finding points with the above algorithm. After the algorithm has found a solution, we are going to run it again, with the same pheromone matrix, but with updated values, where every pheromone in the trails evaporates a small percentage before each iteration. A timer is being set, which controls for how long the algorithm runs.

In Figure 1 a drawing has been created to visualize the idea of the algorithm:

Figure 1: Illustration of the ACO algorithm, showing how it can be used, and how it process next moves
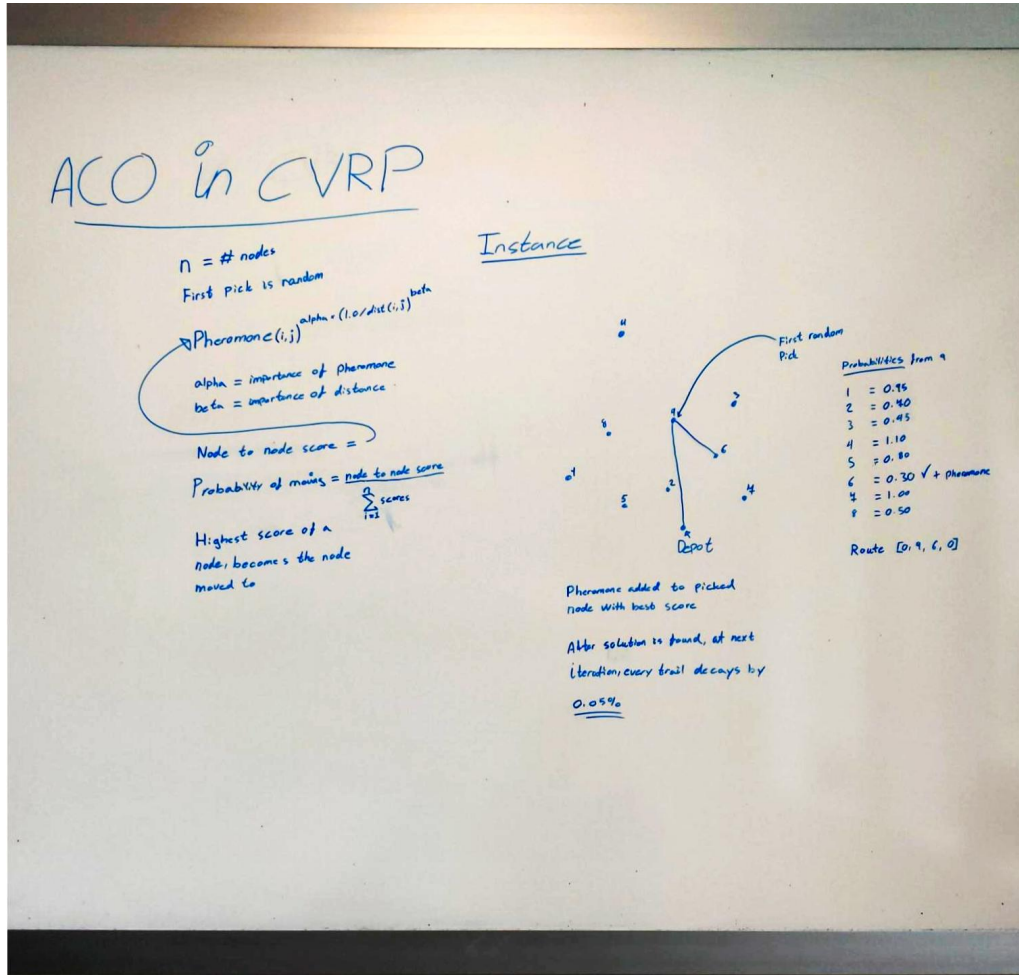
The complexity of the algorithm is $O(n)$ after initializing the pheromone matrix, due to the amount of checks from the current point it has to perform, which is n

The psuedocode of the algorithm can be found in the appendices.

4 Discussion

Metaheuristics takes longer to execute, since they are not stopping until certain criteria is fulfilled. This criteria can be hard to reach and therefore, the algorithm can be running forever. In our two metaheuristic algorithms, the stopping criteria

is the timespan of the running process. When the algorithms have been running for the set time, the processís stopped.

Our Simulated Annealing algorithm does run for a long time, but finds good solutions

5
FIGURES

Appendices

A   Metaheuristic - Ant Colony Optimization - Pseudocode

Algorithm 4 Metaheuristic - Ant Colony Optimization

1: `global_best_route` ← empty route/array
2: **function** ALGORITHM
3:   **while** A maximum number of iterations or a time limit has passed **do**
4:       `local_best_route` ← empty route/array
5:       `current_route` ← empty route/array
6:       `capacity` ← 0
7:       `alpha` ← A random number
8:       `beta` ← A random number greater than alpha
9:       `evaporation` ← Some small number
10:       `pheromone_matrix` ← Contains every pheromone for each pair
11:       `sum_of_all_pheromone` ← The summation of the pheromone_matrix
12:       **for** $first\_point \leftarrow 0$ to length of data **do**
13:           **for** $second\_point \leftarrow 0$ to length of data **do**
14:               **if** $first\_point \neq second\_point$ **then**
15:                   `pheromone_matrix(first, second)` ← $1/distance$
16:                   `sum_of_all_pheromone` ← append calculated pheromone to the sum
17:       Pick a random point for the list of points.
18:       Mark it as visited
19:       Append it to the routes
20:       Keep storage of the previous score
21:       **while** $point \neq$ visited in list **do**
22:           **for** key in pheromone_matrix **do**
23:               Calculate and store the Score-equation
24:               Calculate and store the Probability-equation
25:               **if** the new score previous score && key is not visited **then**
26:                   Set the key of the request to the point moved to
27:           Append Newpheromone-equation to the trail of the point moved to
28:           **if** Capacity of picked point $\leq CapacityConstraint$ **then**
29:               Capacity capacity of request
30:               Append picked point to `current_route`
31:               Mark picked point as visited
32:               Set picked point to the current point
33:               Set previous score to 0
34:           **else**
35:               Reset `current_route` and append picked point to new route
36:               Mark point as visited
37:               `capacity` ← 0
38:               `capacity` ← $capacity of picked point$
39:               Set previous score to 0
40:       **if** `current_route` is better than `local_best_route` **then**
41:           `local_best_route` ← `current_route`
42:       **if** `local_best_route` is better than `global_best_route` **then**
43:           `global_best_route` ← `local_best_route`
44:       loop through `pheromone_matrix` and evaporate every pheromone trail by a small percentage

---